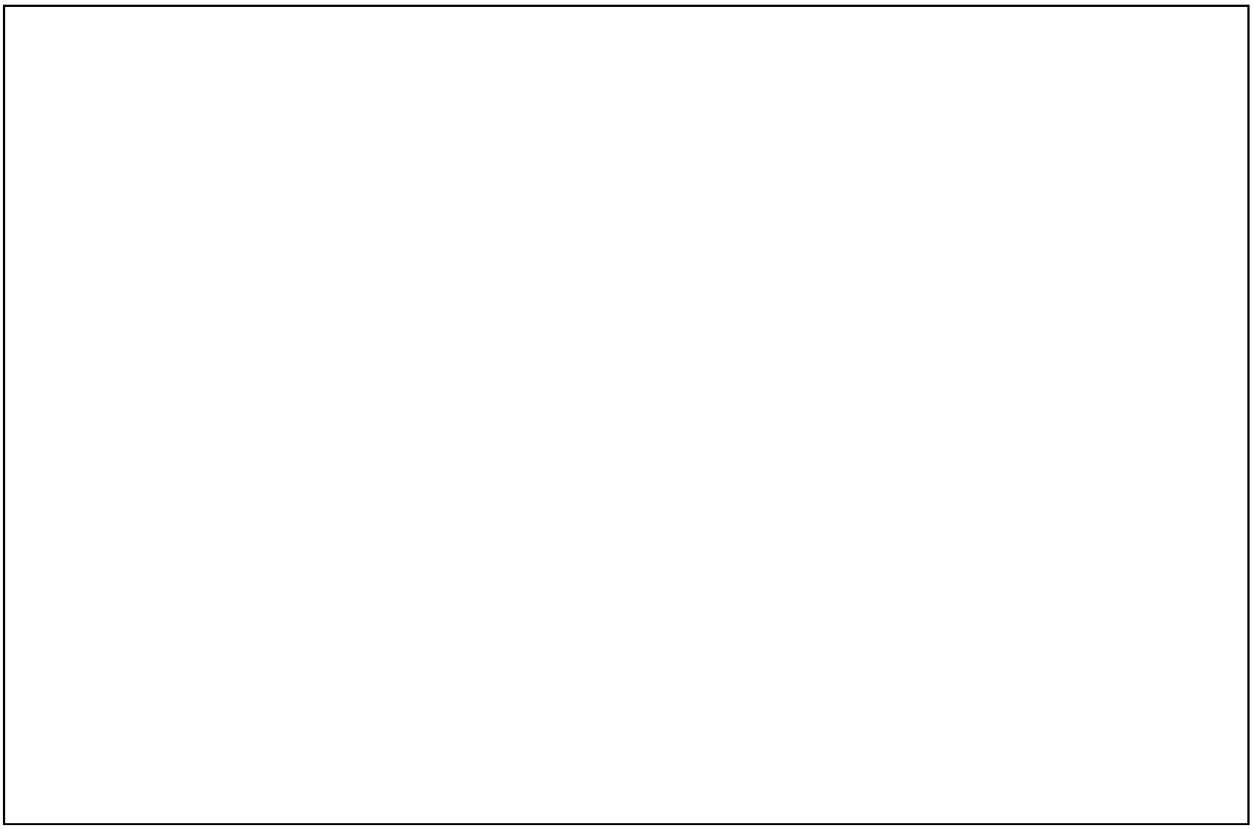
ERICSSON

Tasks for ELTE Testing Telecommunication Systems Laboratory

Project Lead: Prof. Attila Kovács

Technical Lead: Kristóf Szabados, PhD.



Contents

1 Features with high priority

2 Bugs

2.1 Extract definition problem

The extrect function refectoring is not copying the module's with attributes to the new project.

3 Java code generator related

3.1 Checking new features of Java to speed up Titan

(Should be simple after selecting a feature that can be used)

The task here is to select a feature from the newer Java versions (>=1.8) and check manually, what kind of effect it would have on the generated code, if we would update our Java code generator. Demonstrated with example codes.

3.2 Why is Java so fast?

(can be hard, needs background in Java)

While rewriting the C++ runtime to a Java version, we have noticed that Java in many cases is much faster … in some extreme cases it might perform bit operation magnitudes faster.

The task is to show how this is possible in detail. This will involve many measurement, and a good understanding of Java generated code (to understand how it changes during execution).

4 Database related

4.1 Efficient web interface for database of usage data

The task here is to create an efficient way of visualizing/handling data from a large database (100 thousands or millions of records), so that it does not kill a modern web browser to display usefull information, and navigate it.

In this task we provide the information on what we are interested in and how it should be shown.
Preferably should be done in PHP.

4.2 Infering information from database of Usage data

The task here is to infer interesting information from a large database (100 thousands or millions of records) holding usage information.

In this task we might not need web interface for the solution, the task is focused on analysing the data, to detect trends and other information that can be interesting.

5 Manual Test Structure refactoring

5.1 Re-modularization of a test system for more efficient build performance in Continuous Integration.

(simple, could be done by several)

The task here is to study the re-modularization of an existing test system so that it fits better to current environments, based on performance measurements.

When initially designed, hardware were mostly single core, singe thread achines, for which sequential test compilation and execution was optimal.
But nowadays most servers and laptops have massive amounts of CPU cores, that could make it possible to compile several source files at the same time.

This task is main manual, but also requires measurements to prove refactoring steps.
For example: modules containing commonly used types and functions were used to reduce the complexity of the actual modules holding tests and speed up compilation. In a parallel build situation any unnecessary dependency between files should be theoretically removed as it hinders parallelisation, yet if too large parts of the common codebase need to be copied into the modules holding tests, the increase in their size might actually result in the build process finishing later.

6 Designer features

6.1 Searching for write locations

(Semantic tree, TTCN-3 language)

Implementing the usual F5 feature for the Designer (find writes).

This is a feature similar to finding all references (F4), but we are only interested in results where the variable is written, or used as actual parameter to an out/inout formal parameter in a function call.

This feature can simplify the debugging process.

6.2 Re-thinking the interaction between the outline view and the AST.

(Semantic tree, pattern/design task)

Currently the outline view is operated by functions present in the semantic trees classes.

The task is to check/analyze if this featue could be implemented using the Visitor pattern. If the modification is benfitial it also has to be done, if it is not possible or not worth it proving this condition is needed as the solution to the task.

6.3 Re-thinking the interaction between Find proposal and the AST.

(Semantic tree, pattern/design task)

Currently code completion is using special functions embedded into the AST for its operation.

The task is to check/analyze if the part of code completion where the sopces are searched could be implemented using the Visitor pattern. If the modification is benfitial it also has to be done, if it is not possible or not worth it proving this condition is needed as the solution to the task.

6.4 Re-thinking the interaction between code completion and the AST

(Semantic tree, pattern/design task)

Currently code completion is using special functions embedded into the AST for its operation.

The task is to check/analyze if the part of code completion where the reference is traversed from an assignment could be implemented using the Visitor pattern. If the modification is benfitial it also has to be done, if it is not possible or not worth it proving this condition is needed as the solution to the task.

6.5 Showing in the Component hierarchy

(simple, eclipse action)

It would be good if the user could find the location of a component in the component hierarchy starting from the component in the source code.

The function needs to work when started on a component in the source code, open the existing component hierarchy view and select the located component (zooming, coloring, etc..)

6.6 What causes an import cycle?

(Semantic tree)

Find the definitions that cause the cyclic imports between modules.
These are definitions, which are imported through an import taking part in the cycle.

7 Code smell detectors

7.1 Are there types/functions/testcases mixing within a single module?

(simple, introductory task)

There was an old organizational principle to put types in one module, functions in an other, testcases in yet an other, etc…

Although this is considered a deprecated method, it might still be possible to support it with a code smell checker, for those required to use it.

7.2 Report user defined encoding as warning

(Easy, introductory)

Titan does support user defined encodings on types, to provide flexibility. But this also comes with the cost, that when a user misstypes something it can not be reported as an error immediatelly.

The task here is to report as a code smell (with warning severity by default) when the encoding of a type is not one of the encoding provided by Titan, but has a name close to it.

7.3 Bringing template restrictions ahead.

In TTCN-3 it is possible to restrict templates in 3 ways. Increasing the safety of their usage later or bringing their checks forward. For example, it is only allowed to send templates that are values.

Usually these are last minute checks. This could be helped if we culd warn the user where they could use stronger checks.
For example a template to be sent on the network should have (value) restriction … and if another template is to be assigned to this template, it should also have (value) restriction.

For example:
“var template integer xxint2;
Port2[i].send(xxint2);”
In this case it would be valueable that xxint2 should be template (value)

7.4 Select optimization opportunities.

It might be possible to detect what is hindering the code generator to generate more optimal code, for select statements. For example overlaping branch conditions.

7.5 Checking @deterministic

(simple, TTCN-3)

The latest TTCN-3 standard allows the marking of functions as deterministic.

It might be possible to automatically detect the incorrect usage, or missing of this notation.

7.6 Detecting when valueof expression is used to assign a value to a template.

(Simple, introductory task)

It is useless to assign a template to an other template, and calling the valueof expression in between. In such cases the template is converted to a value and back.

```
„
template Sh_XmlData opCat0Name
:=valueof(t_sh_OperatorCategory(operatorCat0Name));

”
```

Technically this is correct if the valueof expression has as actual parameter a generic function call, but in this case we could warn the user that the function should return (value) restricted template, for stricter checks.

7.7 Using unbound fields as optional.

Optional fields of records can only be used safely, if they are checked to be present in advance.

The task is to find the locations, where this was not done.

7.8 Using an unkonwn union branch.

It is only safe to refer to the contents of a union’s branch, if it was previously checked to be active.

The task is to find location, where this was not done.

7.9 Indexing with formal parameters without checking

When a formal parameter is used to index a list, it is important to check if it is withint the valid range.

The tastk is to find locations where this was not done.

7.10 Using type compatibility.

The TTCN-3 language allows for the users when they assign a value or call a function, to use a variable who's type is only compatibly with the expected one.

This is for convenience, as the code does not need to have that attention, but this results in a runtime cost. There must be a conversion between the two types.

Hint: „isCompatible" vs. „isIdentical" checks.

7.11 Module partitioning

(hard)

For large modules it is a valid question if they can be partitioned into smaller ones, and if yes … how.
Technically the interesting partitionings might reduce the interfaces, or help speed up compilation.

7.12 Reading out parameters, before writing inside conditional brances.

(hard)

When a function is called the value of it out formal parameter is set to unbound automatically. If this formal parameter is read before a value is assigned to it, that will result in an error and ending the testcase.

Titanium can already detect if the out parameter is read before written in simple branchless code.
The task here is to extend this feature with supporting the detection of reads before writing in conditional branches.

7.13 Reading a local variable before writing.

(hard, has example)

In theory this is the same task as in the case of out formal parameters: variables first need to have a value assigned to them before they can be read.

The task is detecting such cases.

7.14 Common subexpression detection

(hard, Semantic tree)

It does happen that in the same function a complex expression appears several times, that could be extracted into a local variable.

The task is to detect such constructs.

7.15 Out formal parameter only written conditionally

(hard)

When a function is called the value assigned to out formal parameters is set to unbound automatically.

Titanium can already detect if the out formal parameter is read before written in branchless code.
This task is about detecting execution paths in the functions, that can leave out formal parameters without being written to.

7.16 Overlapping select branches.

(Moderatly hard, Semantic tree)

It can happen the ther is an overlap among the conditions of the select branches. This result is the first ocasion „hiding” later branches, that will never be executed. On the other hand as this might require checks with large computational cost, it should not be part of the base semantic checks.

The task here is to collect the conditions of select branches and check/determine if the same value can fit more than one branch.
Care has to be taken with complexity: it is enough if it works for numbers, and only later gets extended to complex structures.

8 Visualization

8.1 Configuration extraction

One method to extract the component configuration:
- the components are boxes
- The connect and map statements connecting them.

This should work on testcases, by analyzing the statically reachable functions from each testcase.

But actual visualization is not important, it enough to save the data into a proper format.

8.2 Radial DAG Display

(simple, graphics)

In the current architecture visualization it is possible to extract the module dependency graph and create a directed acyclic graph representation of it, that orders the nodes into „layers”.

The current task is to take this data/information structure and diplay it in a circular way. Each „layer” being a circle around middle point. The distance of the „layers” from the middle point calculated from the layer numbers, and the actual arc where an actual node is calculated by deviding the arc with the number of points ont hat „layer”.

Please note that this is an early experimental visualization. It might not turn out to be the most usefull for the first try, but can be improved further.
With the aim being to check if such a display could provide us with some interesting insight ont he architecture of a project.

8.3 Helping windows to show directions.

(simple, graphics)

Some people have notifed us, that it would be nice for newcommers if there would be a windos presenting the controls when the graph displaying views are first shown.

As getting to used to the controls is 3-4 minutes, this task is not high priority.

8.4 Visualizing project structure

The architecture visualization of Titanium can show module and component hierarchy.

This task is about creating a visualization that displays the hierarchy of prejects depending on each other.

9 Automated Refactoring Tools

9.1 Canonizing partially initialized values.

In TTCN-3 it is possible to leave out optional fields during initialization (but instead use the implicit attribute to fill it automatically).

The task here is a refactoring that would explicitely write in the implicitly missing fields.

9.2 Making values with lots of omit fields implicit.

(easy, Semantic tree)

The opposite of the previous refactoring, to reduce the textual size of the values if needed.

9.3 Collecting anytype opportunities

Currently, in Titan, users have to list the types that can be use in the anytype of a module, at the end of the module in an extension.
And it might happen, that they forget about some.

It might be interesting to create a refactoring that finds all types in a module, and extend the list of anytype elements automatically.

+ maybe alphabetic ordering.

9.4 Mergeing assignment sequences in large scale.

(hard)

Can we detect and improve on a large scale when the user has consecutive assignments to the same value?

9.5 Canonizing function calls with named assignments.

(Simple, Semantic tree)

In TTCN-3 it is possible to provide the actual parameters of a function call in any order, by telling which actual parameter belongs to which formal one.

This refactoring should canonice such calls, by putting the assignment in order and maybe filling in missing places with the default actual parameters.

9.6 Shortening of canonical function calls

(hard, might not be solvable)

The revers of the previous, where the function call is shortened leaving out values that are identical to the formal parameter's default value.

9.7 Writing out @deterministic where possible.

(connecting to 7.5)

@deterministic is a new language feature, which is rearely present in the code for this reason.

Users might be interested in it to make stricter restrictions.

9.8 Recuding the scope of local variables.

(hard, partially done)

In case a variable is declared on function scope, but only used within a conditional branch, it should be moved into the branch.

9.9 Changing sizeofs in for loops.

Checking sizeof of a variable in a loop, whithin which its size does not change is wasting time.

These case be refactored automatically, so that calculating the sizeof happens once and gets saved into a variables. In these cases the loop condition only needs to use this variable.

9.10 Extracting a definition into a new project from the control part.

(Standardizers would like this)

The already exsiting definition extraction could be extended to support extraction from control parts too.

9.11 Unused global -> unused local

(Simple)

Unused Global constants can not be deleted from the code, as it is not known if a user is using it or not.
But they can be turned private or moved into a local scope … and if noone has a problem with this for a long time it can be deleted.

The task here is to „hide” unuse global definitions in the above mentioned ways when possible.

9.12 Common subexpression extraction

(hard, can be a research topic)

Expression that occure several times withint the same function, might be automatically extractable to a single place.

The task here is to do these extraction. Care must be taken as the detecting is an other task waiting to be implemented.

9.13 Supporting Module partitioning

(hard, can be research topic)

The task is to find refactoring methods, that are able to automatically partition modules into several new modules.

It might help if, based on the inward references, we can identify what does not belong there.

Partitioning functions can be a good starting direction too.
Just like extracting select branches into functions using „handle_event” like naming conventions.

9.14 Merging if branches

In some cases conditional checkes embedded into other conditional statments can be merged.

9.15 Removal of unreachable code.

The task is to create a refactoring method that can automatically remove unreachable statements.

9.16 Code formatting on semantic base

(hard and large task)

The task is a refactoring feature that is able format source code based on semantical structure and a given set of configuration options.

This could be an advanced version of the current indentation correction feature.

9.17 What other automated refactoring could we support?

For example a set of ideas: http://autorefactor.org/html/samples.html

10 Other large tasks

10.1 Comments should belong to their definitions.

(hard, important)

A peculiar problem of the current code is that comments are not correctly matched up to definitions, so we can't analyze them correctly.

Most probably the problem stems from the parser detecting definitions, but the lexical analyzer detecting comments … and their ability to not stay in synch.

10.2 Graphical value/template editor.

(hard, design, graphical user interface)

Currently the designer allows only textual value and template editing. This is efficient in writing programs, but might not be the most efficient in understanding and filling in large and complex data structures.

The task is to experiment with a way of creating graphical editing interface for manipulating values and templates.

10.3 Analyzing configuration files semantically.

(hard, large, design task, Semantic tree)

Currently our IDE is able to analyze ASN.1 and TTCN-3 source code. Plus configuration files can be syntacitcally analyzed, but not semantically.

The task is to build a structure similar to the designer's AST, with the configuration file's parser.

10.4 Paths between 2 modules.

(graph theory, graphics)

The task is to extends Titanium's hierarchy viewer with the ability to display all possible paths between two selected modules, in the graph.

10.5 Analyzing Designer's or Titan's tests with XCoverage

The Designer's tests were designed to maximise coverage.

The task here is analyze the coverage with XCoverage and improve by adding tests (and information on the performance of XCoverage).

10.6 Checking the new features of C++ ’11, ’14 and ’17 to speed up Titan.

The task here is to select a feature from the newer C++ standards and check manually, what kind of effect it would have on the generated code, if we would update our code generator. Demonstrated with example codes.

(rvalue -s are already done)

10.7 Optimizing the initialization of complex strustures in the generated C++ code.

(C++)

In the case of complex type, which have all of their fields initialized it would be possible to generate a different code.

The task would to understand the semantic analysis and code generation of Titan and check what effect it would have if we optimized the generation of such cases.

10.8 Constant folding for templates

(C++)

Titan currently is able to do constant folding on constant values.

The task here is to implement a similar feature for templates.

10.9 Tutorials with the help system of Eclipse

(new)

The help system of Eclipse, in theory, provides us with opportunity to create feature specific tutorials. Where users can learn and practice the use of some of the features.

The task is analysing these features and creating a few such tutorials.

10.10 Eclipse templates configured by the users.

(new)

Eclipse provides the opportunity for the user to provide his/her own template on workbench level. Which can later be used in code completion.

The task is to analyze this feature and create an example implementation.

10.11 Writing a new Open Source ANTLR plugin.

(a new plugin)

The current Open Source plugin for ANTLR has several issues.
The bigest being its reliance on XTEXT, which has to be installed for it to work … but only a given version, which is no longer easily reachable, and can not be updated.

An other issue is that the editing functionality requires too large computational capacity.
And finally the plugin is not able to handle dependency between language files.

The task is creating a new plugin, that does not use XTEXT, does not provide its own editor, but can handle dependencies between grammar files (even if it has to be explicitly added by the users).

10.12 Code completion with multiple pages.

(close to the platform, hard to guess complexity)

Currently the Designer offers all code completion candidates it found in a single list.

The task is to turn in into a multy-page offering, similarly to how it done in JDT.

10.13 Syntax highlight based on semantic information.

FIXME: details

10.14 Extending constant folding

(hard task, but could be paritioned)

The constant folding of Titan could be extended so that more and more non-constant parts of Titan's regression tests could get evaluated in compilation time.

This could actually be several tasks, depending on how many opportunities we can find in the regression tests.

11 External/Research topics

They are rather interesting topics, if someone would like to research in their direction.

11.1 Tool support memory lead detection.

Gabor Toth A. is the contact.
The aim is a tool that is able to help detecting memory leaks, based on few and limited outputs.

11.2 Checking Sonar Qube metrics

The task here is to check what exactly are Sonar Qube metrics measuring precisely. There is a good chance, they are actually measuring something else than what their name suggests.

11.3 Is it meaningful to measure McCabe on testcase level?

In theory it might be usefull information to know the complexity of the whole systems.

Cuting a complex function in 2 results in 2 functions with smaller complexity each … but does not decrase the overall complexity of the whole system.

11.4 Can we use statistic distribution to tell if something at the wrong place?

Comptemporary software technology knowns that in programs accepted as „good quality" the number and size of functions follows a power law distribution. And also Les Hatton has shown that this has a background in information theory.

So what would happen if we accepted these ideas, and tried to detect issues based on them? (here the relationship changes and we would use the distrubution breaking places to find problems)

11.5 Can we better detect groups in XCoverage matrices?

Currently the groups in the XCoverage matrix are identified based on static boundaries, and global differences.

Can we find a better algorithm, that can use local differentiations too, to find better groups?
For example: first identify 2 groups based on a separation line, than identify a line for each of them, creating hopefully even more connected components. Using only 3 lines.

11.6 Dataflow analysis on TTCN-3 code.

The task is to simulate the execution of the code statically, so that runtime errors could be detected earlier.

If we could use every testcase as an entry point to start a simulation, track the values of local variables, we could find interesting code parts in smaller source codes (meaning errors, optimization opportunities, access issues, etc..).

11.7 Connecting Code smell –s to people.

Are there personal trates for creating some code smells?

Currently we can identify several code smells with their locations. If we could collect who wrote/modified those lines, we could connect code smells to people. In this way maybe we could tell if some people are more prone to create specific code smells, than others.

11.8 Keywords: „Propagation Cost” and „Dependency Density”

The matrix of XCoverage could eb understood as a dependency matrix of a graph if we can find good limits based on similarity. One we have that we could check if some graph measurements could help us uncover interesting findings.

11.9 Can we optimize the compiler/Designer generated code further?

There some more optimization opportunities for generating more efficient code.

Constant folding working on templates, optimized initialization of structured values, etc...

11.10 Check the consistency of developer productivity using version control systems.

With the help of gitstat, or the tool of Attila Zsiga we can extract the information of how many commits and code each developer did on a daily bases.

We could check if there some trends related to persons/projects/industries.

11.11 How many are reviewing a commit?

It would be interesting to check a large review database to see how many are reviewing individual changes. For example: on gerrithub, what is the average number, and what is the difference between a change reviewed by 1 person and a change reviewed by many?

11.12 How does instability change during the lifetime of a project?

Instibility is a metric producing values between 0 and 1, showing how instable a part of the arhictecture is. A class not build on anything has 0 (as only its own change can cause problems), a class that builds on other and nothing builds on it has a value of 1 (as changes in many places can affect it, and there is nothing to stabilize it).

While measuring on a large industrial system, we have seen that this function gets shifted as the number of modules increases, resulting in more unstable modules.

Ont he other hand instability might not be a single function.
The task here is to investigate with mathematical methods the possibility that, the instability function is actually an aggregation of some other functions.

11.13 Is it a technical debth if the developer does not know the code (article)?

11.14 How would generating var -s in Java affect compilation and runtime performance (article)?

11.15 Is it possible to make the Elipse IDE more responsive?

11.16 TitanSim trend update (article).

11.17 Titan trends.

11.18 Refactorings of Titanium (article).